Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds

Jad Darrous, Shadi Ibrahim, Amelie Chi Zhou, Christian Pérez

Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds

Jad Darrous*, Shadi Ibrahim†, Amelie Chi Zhou‡, Christian Perez*
* Univ. Lyon, Inria, CNRS, ENS de Lyon, UCBL 1, LIP, Lyon, France
{jad.darrous,christian.perez}@inria.fr
† Inria, IMT Atlantique, LS2N, Nantes, France
shadi.ibrahim@inria.fr
‡ National Engineering Lab for Big Data Computing Technology, Shenzhen University, China
chi.zhou@szu.edu.cn

***Abstract*—Recently, most large cloud providers, like Amazon and Microsoft, replicate their Virtual Machine Images (VMIs) on multiple geographically distributed data centers to offer fast service provisioning. Provisioning a service may require to transfer a VMI over the wide-area network (WAN) and therefore is dictated by the distribution of VMIs and the network bandwidth in-between sites. Nevertheless, existing methods to facilitate VMI management (i.e., retrieving VMIs) overlook network heterogeneity in geo-distributed clouds. In this paper, we design, implement and evaluate Nitro, a novel VMI management system that helps to minimize the transfer time of VMIs over a heterogeneous WAN. To achieve this goal, Nitro incorporates two complementary features. First, it makes use of deduplication to reduce the amount of data which will be transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. Experimental results show that our network-aware data transfer strategy offers the optimal solution when acquiring VMIs while introducing minimal overhead. Moreover, Nitro outperforms state-of-the-art VMI storage systems (e.g., OpenStack Swift) by up to 77%.**

***Index Terms*—Geo-distribution, deduplication, virtual machine image, data transfer, scheduling**

I. Introduction

Nowadays, major cloud providers deploy their services on geo-distributed infrastructures. This worldwide distribution provides low latency for the end users of the services. For example, Amazon EC2 currently has 18 geographically distributed service regions [1], and Windows Azure operates in 36 geographical locations [2]. In general, to build a service, a *Virtual Machine Image* (VMI) is required. Such VMI includes an operating system (OS) and a "service-specific" customized software stack.

It has been pointed out that [3], [4], [5], [6], [7], [8], [9], [10], [11], rather than (re)creating a VMI from a base image (i.e., golden image with a specific OS) when provisioning a service, it may be better to provide a set of customized VMIs and (re)use them when needed. For this reason, cloud providers supply users with a number of VMIs to facilitate their service provisioning. However, the huge number of possible combinations of different operating systems and software stacks leads to a continuous increase in the number of VMIs, e.g., Amazon EC2 provides more than 19,000 public Amazon Machine Images (AMIs)[1].

VMI management has therefore become an important issue in clouds. Prior literature has mainly focused on leveraging *deduplication techniques* to eliminate redundant blocks in-between VMIs and therefore reduce the storage space within a *single* data center [6], [7], [8], [10]. However, few works have explored VMI management in geo-distributed clouds. The geographical spread of data centers, the continuous increase of VMIs number, in addition to the limited bandwidth and heterogeneity of the WAN connection, have elevated VMI management to a key issue in geo-distributed clouds. Nevertheless, several major challenges arise when dealing with geo-distributed VMIs:

- **Challenge 1**: The size of VMIs is critical for fast service provisioning, as the size of a single VMI can reach dozens of GBs[2]. Previous efforts try to reduce VMI size by exploiting similarities within and in-between VMIs, through deduplication, to reduce the storage cost. However, they do not evaluate the impact of deduplication on the service provisioning time, when the VMI is pulled from different geo-distributed sites over high latency and low bandwidth WAN.
- **Challenge 2**: On the one hand, it is not practical to replicate VMIs on *all* sites. The cost, in term of data transfer across data centers, may become prohibitive as the size and number of VMIs increase. For example, maintaining one single VMI introduces high transfer cost when it is updated frequently; security patches alone may result in almost 150 updates per week [12]. On the other hand, replicating VMIs on (few) geo-distributed sites to ensure high availability and meet users' needs poses a challenging issue when provisioning VMs, especially as the large size VMIs must be pulled – over WAN – from multiple geo-distributed locations.
- **Challenge 3**: Previous solutions which adopted deduplication techniques assume a constant cost when retrieving

[1]We obtained the number of AMIs from Amazon EC2 Dashboard on November 16^{th}, 2017.

[2]http://dash.vopendata.org

the VMI chunks and thus use simple and random order. They will result in long provisioning time when applied directly in geo-distributed clouds, due to the link heterogeneity. For example, the bandwidth in-between 11 sites in Amazon EC2 varies by up to 12X [13]. An optimal retrieving plan is therefore imperative to improve the provisioning time. However, finding the optimal solution comes with a high computation overhead due to the huge number of chunks. For example, as shown in Section VII, it takes almost 267 *s* to find the optimal plan to pull 10,000 *chunks* from 4 *sites*.

Contributions. To address the challenges above, in this paper, we design and implement Nitro, a novel VMI management system for geo-distributed clouds. Unlike previous VMI management systems, which focus on facilitating VM provisioning within a data center (i.e., optimizing VMI transfer from storage nodes to compute nodes) [4], [5], [6], [9], [10], [11], Nitro focuses on minimizing the transfer time of VMIs over a heterogeneous WAN which is the main performance bottleneck when provisioning services in geo-distributed clouds. To achieve this goal, Nitro incorporates two complementary features. First, it makes use of deduplication to reduce the amount of data which will be transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. The network-aware strategy embraces an algorithm that produces an optimal chunk scheduling in polynomial time based on graph flow algorithm. To reduce the overhead and improve the scalability of our designed algorithm, we propose a grouping optimization to reduce the number of chunks in the graph. We have implemented Nitro – and used Redis [14] as a backend storage – with intensive evaluations on Grid'5000 [15] – an academic experimental testbed based in France. Experimental results show that the network-aware data transfer strategy offers the optimal solution when acquiring VMIs while introducing minimal overhead. Moreover, Nitro outperforms state-of-the-art VMI storage systems (e.g., OpenStack Swift) by up to 77%.

The remainder of this paper is organized as follow. Section II introduces the related works about VMI management. In Section IV, we explain the network-aware chunk scheduling algorithm. Section V presents Nitro and its workflow. Experiment methodology and results are discussed in Sections VI and VII. Finally, Section VIII concludes this study.

II. Related Work

VMIs are usually stored on storage nodes. Upon a user request to provision a VM, the specified image is transferred to the local disks of the compute nodes. Thus, a huge amount of data is eventually moved over the network consuming network bandwidth and creating network overhead. This network overhead plays an important role on the VM provisioning time in a single data center. Importantly, the impact of the network on provisioning time is amplified in geo-distributed clouds because of the high latency and the limited bandwidth in WAN. Hereafter, we discuss VMI management in data centers and present current approaches and systems that facilitate data access in geo-distributed clouds.

Leveraging deduplication techniques for VMI management. Due to its wide adaptation in archiving systems [16], deduplication techniques have been extensively studied for VMI management. The huge size of VMIs and the high similarities among them are the main motivations behind these works. Deduplication techniques can be applied on the *file-level* or *block-level*. File-level deduplication eliminates duplicated files in the file system of the image. In contrast, block-level deduplication treats the image as a raw array of bytes and eliminates duplicated segments of bytes. The majority of works on VMI management employ deduplication on the *block-level*. Previous works [3], [4] show that fixed-size chunking can achieve high compression ratio, up to 80%, yet it is as good as variable-size chunking and sometimes outperforms it.

In addition to storage reduction, deduplication techniques improve the provisioning time of VMs by exploiting similarities between chunks of the requested VMIs and the VMs running on the compute node (i.e., host machine) [6], [7], [8], [9], [10]. Nicolae et al. [11] consider VMIs deduplication and introduce a P2P collaborative chunk sharing between the compute nodes to reduce the contention on storage nodes in case of multi-deployment (i.e., requesting the same VMI by multiple compute nodes) and therefore reduce the provisioning time. Although the aforementioned works can improve the provisioning time but they are centered on VMI management inside a *single* data center, where the main focus is how to efficiently broadcast VMIs from the storage nodes to the compute nodes. In contrast, Nitro leverages deduplication to reduce the amount of data transfer in-between geo-distributed sites. Furthermore, Nitro carefully pulls chunks from different sites to reduce the provisioning time. Actually, Nitro complements these works to further improve their performance in geo-distributed environments.

A closely related work is Karve et al.'s proposal for leveraging data deduplication for VMIs in geo-distributed data centers [5]. The master site maintains the global view of the system and the distribution of the chunks, thus it is responsible for creating the transfer plan of chunks to the destination site. This work is limited to small scale systems and may suffer noticeable performance degradation due to the centralized management overhead when dealing with huge number of VMIs. In addition, it does not consider network heterogeneity between data centers which we are targeting in our system. In Nitro, we opt for a decentralized VMI management and propose to group chunks into mega-chunks to further reduce the overhead of finding the optimal chunk transfer plan.

Storage solutions for VMIs in data centers. VMI management in data center is an issue of high importance. Traditional distributed storage systems (e.g., HDFS [17], Ceph [18], etc) can be used to store and manage VMIs as they provide high storage capacity. However, they are optimized to handle data read and write within a data center. OpenStack Swift is an

open-source distributed object store [19]. Swift is recently advocated as the *de-facto* industrial storage system for VMIs. It supports geographically distributed clusters, thanks to the read and write affinity properties: regions can be statically prioritized to favor read and write from/to "nearby" sites (i.e., normally sites that have a higher bandwidth between them). Prioritizing "nearby" sites may stress the network links to these sites and thus may result in longer transfer time. In contrast, Nitro pulls chunks from all sites considering both network heterogeneity and balanced load distribution between sites. InterPlanetary File System (IPFS) [20] is a peer-to-peer distributed file system that is designed for massively distributed environments. IPFS stores blocks of data indexed by their fingerprint (i.e., cryptographic hashes) and therefore is able to perform deduplication at the complete file system level. Furthermore, IPFS exchanges chunks using a BitTorrent [21] inspired protocol, named BitSwap. However, to reduce the complexity of finding the optimal plan to pull a file – due to the huge number of replicated chunks – IPFS simply pulls chunks from all available sites and therefore introduce a huge network overhead. Finally, content delivery systems like BitTorrent [21] can be also used for VMI distribution, but it does not take into account network heterogeneity when retrieving data as peers are selected randomly. Alternatively, it addresses network heterogeneity by pulling chunks from multiple peers to avoid the impact of weak links. This may lead to good performance but at the cost of high network overhead.

Data access in geo-distributed environments. Researchers have studied various aspects of data management in geo-distributed environments. Some research efforts have been dedicated to reduce the latency when accessing data in geo-distributed storage systems by migrating data close to the clients [22], leveraging additional storage tiers (e.g., caches) to store replicas of hot data [23], and relaxing the consistency requirements [24], [25], [26]. Performing data analysis on geo-distributed data has been extensively discussed in recent years [27], [28], [13], [29]. These works focus on exploring the trade-offs between utility [13], data locality [28], and performance [29] when moving data to compute nodes. In contrast, we focus on VMI management: we aim at minimizing the time to transfer VMIs between data centers through leveraging data similarity and exploiting links with high bandwidth.

III. Nitro: Design Principles

Nitro is a novel VMI management system responsible for storing and retrieving VMIs to and from different sites in geo-distributed data centers. This section presents the design principles of Nitro, while the following two sections focus on the network-aware VMI pulling and the implementation details of Nitro. Nitro is designed with the following goals in mind:

- **Reduce network overhead:** This is critical in geo-distributed clouds when transferring the large size VMIs. Previous works [3], [4] show that exploiting similarities within and in-between VMIs may result in reduction in storage space by up to 80%. Therefore, Nitro aims at reducing the amount of data transferred over WAN by leveraging deduplication. This will not only reduce the size of acquired VMIs but also effectively increase locally available chunks (on destination site).
- **Network-aware data retrieval:** The bandwidth and latency in-between data centers vary significantly [13]. For that reason, Nitro employs a network-aware chunk scheduling algorithm to find the optimal plan when acquiring chunks from different sites. Thus we can effectively leverage links with high bandwidth and reduce the number of chunks retrieved over weak links.
- **Minimize provisioning time:** Through reducing the amount of transferred data, exploiting chunks locality, and carefully pulling chunks from different sites, Nitro can minimize the transfer time and thus improve the provisioning time.
- **Ensure minimal runtime overhead:** Finding the optimal plan to pull a VMI – due to the huge number of replicated chunks – comes with a high computation overhead. Therefore, we propose a grouping optimization to reduce the problem size (i.e., reduce the number of chunks) when finding the optimal solution. This optimization allows our scheduling algorithm to run in sub-second.
- **Storage backend independent:** Since Nitro is implemented as a separate layer on the top of the cloud storage system, it does not impose any modifications to the cloud system code. Consequently, Nitro can use any storage systems (i.e., key/value store) to store the chunks. This modularity is important as new emerging key/value storage systems can be used to further improve the provisioning time through optimizing data transfer within a data center.

IV. Network-aware Chunk Scheduling Algorithm

This section presents the network-aware chunk scheduling algorithm used in Nitro. This algorithm produces an optimal chunk scheduling that minimizes the transfer time of chunks over heterogeneous networks. It is based on min-cost max-flow graph algorithm and it has a polynomial time complexity.

A. Problem Definition

Consider the scenario of I VMIs and a geo-distributed cloud composed of N sites. Each image is divided into C *equal* chunks and the chunks can be spread to any of the N cloud sites. We suppose that each pair of sites is connected with a dedicated link [30]. When a VMI is requested from a site, we first look for the chunks of the VMI in the local site. If there are not enough chunks to reconstruct the VMI locally, we need to decide which sites to pull the missing chunks from (i.e., the chunk scheduling problem). Our goal is to minimize the time needed to pull all missing chunks from remote sites.

We formally model the chunk scheduling problem using a bipartite graph $G = (V, E)$. The vertex set V includes two types of vertices, namely the set of requested image chunks and the set of all sites. E is the set of directional edges from the chunk nodes to site nodes. An edge from a chunk c to a site s represents that there is a copy of chunk c on site s.

Under the bipartite graph model, the chunk scheduling problem can be described as finding an assignment from chunk nodes to site nodes that reduces the transfer time. As the number of chunks assigned to a site represents the relative time needed to complete the transfer from that site (assuming equal chunk size and homogeneous bandwidth). Therefore, as the transfers can be done in parallel from different sites, minimizing the total transfer time can be done by minimizing the maximum transfer time from each site. Assignment problems in bipartite graphs are often solved using network min-cost max-flow algorithm. Flow algorithm has been used in literature to optimize job placement in data centers [31], [32]. However, classical matching algorithms try to find the maximum match regardless of the mapping. In the following, we introduce the basics of the algorithm and how we have adapted it to solve our problem.

B. Maximum-flow Algorithm

The maximum flow algorithm tries to find the maximum flow that can go through the network from the source node to the sink node, respecting the following two conditions:

- **(1)**: For each edge, the flow going through the edge should not exceed its capacity.

$$F(e) \leq C(e), \forall e \in E$$

  where F and C are the flow and capacity of edge e, respectively.
- **(2)**: For each vertex, the incoming flow should be equal to the outgoing flow.

$$\sum_{u} F(e_{uv}) = \sum_{k} F(e_{vk}), \forall v \in V$$

  where u is an incoming neighbor of v and k is an outgoing neighbor of v.

In our problem, the capacity of edges between the source node and the chunk nodes is 1. The capacity of the edge connecting a site node with the sink node represents the maximum chunks that can be pulled from the site. Initially, it can be equal to the total number of chunks. However, these capacities will change during the execution of the scheduling algorithm. Figure 1 shows the graph representation of a simple example where 5 chunks spread over 3 sites are required.

Directly applying the maximum flow algorithm to our problem using the above graph representation might not generate the edge assignment that we are looking for. This is because the algorithm always tries to maximize the flow regardless to which sites these chunks are assigned to. However, in our problem, it is important to study how the maximum flow is distributed among edges. We design our chunk scheduling algorithm based on this observation.

C. Chunk Scheduling Algorithm

As our goal is to find the chunk assignment solution which minimizes the time needed to acquire all requested chunks, we design a max-flow based algorithm to find the maximum flow solution that also provides a balanced load

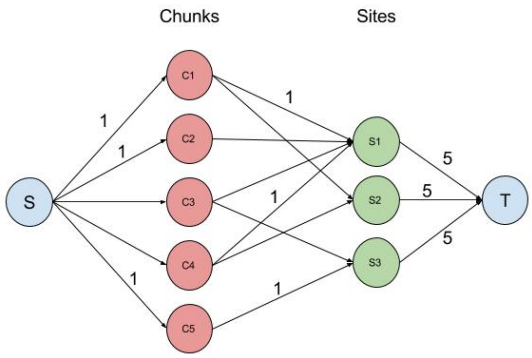


Fig. 1. Graph example: 5 chunks which are spread over 3 sites are required. Initial edges capacities are also shown.

distribution between sites as much as possible. For simplicity, we first assume the network bandwidths between all sites are homogeneous and discuss how to extend our algorithm to heterogeneous network environment later.

As described in the previous subsection, the capacity of an edge represents the maximum amount of flow that can go through it. In our case, the amount of flow going through edges connecting site nodes with the sink node represent the number of chunks assigned to each site, therefore the data size. As the bandwidth is homogeneous between sites, the edge flow actually represents the transfer time. So, minimizing the flow is actually minimizing the transfer time from all sites. Consequently, the idea behind chunk load balancing is how to control the *capacity* of edges connecting the sites with the sink node. Intuitively, a capacity equal to or greater than the number of chunks can always guarantee a max flow solution, whereas a capacity less than $\frac{|C|}{|S|}$ cannot lead to a feasible solution because some chunks remain unassigned. Thus, our goal is simply to find the minimum capacity that can provide a max flow solution in the range $[\frac{|C|}{|S|}, |C|]$. By searching for the minimum threshold which represents the maximum number of chunks that can be pulled from each site, we find the solution with most even (i.e., load balanced) chunk distribution.

So far, we have provided a solution for the case of homogeneous WAN bandwidth between sites. However, in reality, the WAN bandwidth between different sites is heterogeneous and assigning an equal number of chunks to each site, if possible, does not minimize the total time of chunk acquisition, as we will spend more time to pull the same amount of data from different sites. To address the heterogeneity issue, we scale the capacities of the edges connecting site nodes with the sink node according to the available bandwidth (between each site and the destination site) before running the flow algorithm and for every iteration of the algorithm. The intuition behind capacity scaling is that the site with higher network bandwidth can pull more chunks within a unit of time than sites with lower bandwidth.

For example, consider that 4 chunks are required and all of them are available on two sites. One of the sites has a network bandwidth of 50 Mb/s to the destination site and the

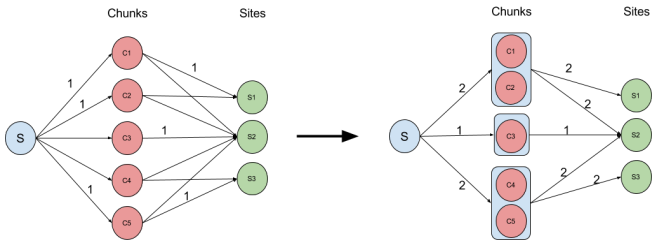


Fig. 2. An example of the grouping optimization.

other site has a bandwidth of 150 Mb/s. Without considering network heterogeneity, the scheduling algorithm will retrieve two chunks from each site. However, the optimal solution would be to retrieve one chunk from the first site and three chunks from the second site.

D. Grouping Optimization

As a VMI can be composed of dozens of thousands of chunks on average, the bipartite graph can become very big. To reduce the overhead and improve the scalability of our designed algorithm, we propose a grouping optimization to reduce the number of chunks in the graph.

The number of chunk nodes can be reduced by grouping the chunks that can be found in the same set of sites into one chunk node, denoted as *Mega Chunk* (MC) node. The number of mega chunk nodes has an upper bound equal to $2^{|sites|} - 1$, which is the cardinality of the set of sites subsets. This upper bound is reached only if there is at least one chunk that is connected to each subset of sites. Also, the number of mega chunk nodes will not exceed the number of chunk nodes, which means that this optimization will be at least as fast as before applying it.

After the grouping, the capacity of edges from the source node to the mega chunks has to be modified according to the sizes of the mega chunks. For example, as shown in Figure 2, the capacity of the edge from the source node to the first mega chunk node is changed to two, as there are two chunks in the mega chunk. A mega chunk can be matched to multiple sites at the same time, which is different from the simple chunk case where each chunk is matched to one and only one site. For example, given a mega chunk node of five chunks, if this mega chunk is matched to two sites, with a flow of two to the first site and three to the second site. As we do not differentiate the chunks, we randomly select two chunks to pull from the first site and three from the second site.

E. Algorithm Overview and Analysis

A summarized pseudo-code of our chunk scheduling algorithm is presented in Algorithm 1. It receives as parameters the requested chunks, the sites to pull from, the current mapping of chunks and the bandwidth scaling (i.e., the relative bandwidths between the destination site and other sites). The algorithm returns an assignment of the requested chunks to the sites. The functions used inside the algorithm are **create_mega_chunks**, **build_bipartite_graph**, **add_capacity_to_sink_edges**, **max_flow** and **remove_set_k**.

- **create_mega_chunks** implements the grouping optimization and groups the input chunks which can be found in the same subset of sites into mega chunks.
- **build_bipartite_graph** builds the graph from the chunks and sites and sets the capacity of the edges as discussed before.
- **add_capacity_to_sink_edges** sets the capacity of the edges linking the site nodes to the sink node.
- **max_flow** is the *preflow-push algorithm* [33] for computing the maximum flow. It takes a graph instance as input and updates the flow of its edges.
- **remove_set_k** is a helper function that removes k elements randomly from the input set and returns them.

Algorithm 1: Network-aware chunk scheduling

```
Input : chunks, sites, chunks_mapping, bw_scaling
Output: chunks_request
mega_chunks ← create_mega_chunks (chunks, chunks_mapping);
G ← build_bipartite_graph (mega_chunks, sites);
low_cap, high_cap ← 1, nb_chunks;
while low_cap < high_cap do
    cap ← (low_cap + high_cap)/2;
    scaled_cap ← cap * bw_scaling;
    add_capacity_to_sink_edges (G, scaled_cap);
    max_flow (G);
    total_flow ← sum_{site=1}^{nb_sites} F(G[site][sink]);
    if total_flow = nb_chunks then
        high_cap ← cap;
    else
        low_cap ← cap + 1;
    end
end
optimal_capacity ← low_cap;
scaled_optimal_capacity ← cap * bw_scaling;
add_capacity_to_sink_edges (G, scaled_optimal_capacity);
max_flow (G);
chunks_request ← {};
for site in sites do
    chunks_request[site] ← ∪ remove_set_k
      (mc, F(G[mc][site])) : for mc ∈ mega_chunks
end
return chunks_request;
```

The algorithm starts by grouping the chunks into mega chunks as we have discussed previously. Then, the graph structure is built from the set of mega chunks and the set of sites. The capacities of the edges are set accordingly, however, the capacities of the edges linking the site nodes to the sink node are left for a later step. The main loop performs a binary search on the capacities range to find the optimal one; in each iteration, we first compute the scaled capacities and then we set the capacities of the sink edges. Later on, we run the flow algorithm to compute the total flow that goes through the network. If the total flow is equal to the number of chunks, that means we can reduce the search range to search for another solution that provides more load balancing. The other case (total flow is less than chunk size) means that not all the chunks are matched and the current solution is not valid, so we increase the lower limit of the search. When the loop is

finished, we run the flow algorithm with the optimal capacity found. At the end, we create the set of chunks that should be requested from each site with respect to the output of the flow algorithm.

The complexity of the algorithm is mainly related to the maximum flow algorithm and the binary search. The complexity of the maximum flow algorithm (i.e., preflow-push algorithm) is $O(|V'|^2 \sqrt{|E'|})$. Where V' and E' are the sets of mega chunks and corresponding edge respectively. The range length of the binary search is $|C|$, which means that the algorithm iterates no more than $log_2(|C|)$ iterations. Putting together the complexities of the two algorithms we find that the complexity of the scheduling algorithm is $O(log_2(|C|) * |V'|^2 \sqrt{|E'|})$.

F. Discussion on Requesting Multiple VMIs

Although our discussion above has been focused on single VMI request, our scheduling algorithm also works when requesting multiple VMIs from the same site. This is because our algorithm is oblivious to the number of VMIs and works on the chunk level. However, if multiple sites are requesting VMIs at the same time, each site will run an instance of the scheduling algorithm separately, and this may cause network contention on the sites that are responding to chunk requests. Addressing this problem and providing load-balance for the multi-sites VMI request scenario is left as future work.

V. Implementation

Nitro consists of roughly 1500 lines of Python code. The current implementation uses Redis [14] as a storage backend to store VMIs, i.e., the chunks of the images. However, any key/value storage system can be used as a storage backend to Nitro. The source code is publicly available at https://gitlab.inria.fr/jdarrous/nitro .

As shown in Figure 3, Nitro consists of two components: a *proxy server* and a *storage engine*. The proxy server is responsible of handling all the requests to VMIs including add, retrieve, etc. The storage backend is where the actual chunks are stored and is implemented as a key/value store. Hereafter, we describe the system workflow.

A. System Workflow

Bootstrapping. To deploy Nitro in a distributed environment, a daemon has to run in each data center. On bootstrapping, these daemons are provided with the endpoint addresses of other daemons.

Adding new image. New VMIs can be added to any site running the Nitro system. The image is split into fixed chunk sizes of 256KB. A list of references to these chunks is also created. This list is used later to reconstruct the original VMI. These chunks are stored into the backend key/value store along with the fingerprint list. The metadata is then propagated synchronously to all other daemons. Then, the universally unique identifier (UUID) of the image is returned to the client indicating the successful termination of the process.

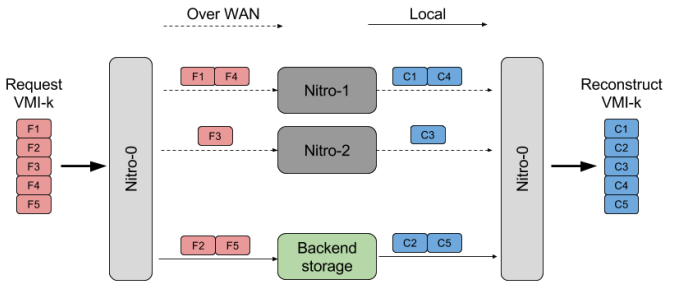


Fig. 3. VM provision Workflow. A VMI with 5 chunks is requested; 2 chunks are available locally and 3 chunks are requested from other data centers. F for $Fingerprint$ and C for $Chunk$.

Retrieving an image. The generated UUID, when adding the VMI, is used later to retrieve it from any other location. Nitro reads the fingerprint list related to that UUID and checks the chunks which are available locally. The list of missing chunks, in addition to the current available bandwidth between the current site and the other sites, are then transferred to the scheduling module that computes the optimal transfer plan to retrieve the chunks from other sites. After receiving all the previously missing chunks, these chunks are stored (i.e., persisted in the Disk, in Redis) and the original VMI is reconstructed and returned to the user.

B. Discussion on Network Performance Variability

Currently, Nitro has no mechanism to deal with the variation of network performance (i.e., bandwidth) during the retrieval of an image. However, the available bandwidth between two sites is relatively stable in the granularity of minutes, as it has been observed in [27], or even 10 minutes [30] which is sufficient to complete a transfer (see Section VII).

C. On Compressing Data Chunk in Nitro

To reduce the size of data transferred over WAN, we further enable data compression on the chunks in Nitro. Chunk compression may result in different chunk sizes, and therefore, may impact the optimality of the scheduling algorithm when retrieving chunks. We discuss the trade-off between data compression and optimal chunks mapping in the evaluation section.

VI. Experimental Methodology

A. Dataset

The dataset consists of 24 VMIs (in raw format) with a size ranging from 2.5GB to 6.5GB for each. As our scheduling algorithm works on the chunk level and is not aware of the total number of images in the system, a dataset of 24 images is sufficient to evaluate the performance of Nitro.

The dataset is built by provisioning the latest versions of eight base images (3 Debian: Wheezy, Jessie, and Stretch; 3 Fedora: Fedora-23, Fedora-24, and Fedora-25; and 2 Ubuntu: Trusty and Xenial) with three software (Apache Cassandra, Apache Hadoop, and LAMP stack) using Vagrant [34].

Table I presents the sizes of our dataset with different storage formats. The *deduplication* and *deduplication with compression* rows represent the cases where each image is deduplicated separately, *without* and *with* further chunk compression, respectively. Whereas the corresponding rows with the *(dataset)* tag represent the case where all the images of the dataset are deduplicated together, i.e., similar chunks of different VMIs are discarded. The *compression* row gives the size of the dataset compressed in `gzip` format. From the presented results, we can notice that applying compression with deduplication (where each image is deduplicated separately) slightly better than compression alone in term of data size reduction. However, Deduplication with compression can by far reduce the size of the data compared to compression alone, as the complete dataset is taken into account. Moreover, the efficiency of deduplication – in contrast to compression – grows when increasing the size of the dataset.

TABLE I
DATASET SIZES UNDER DIFFERENT STORAGE FORMATS

Image storage format	Total size (GB)
raw	104.89
deduplication	50.65
deduplication (dataset)	25.11
compression	19.99
deduplication with compression	17.99
deduplication with compression (dataset)	8.66

B. Testbed

We perform our experiments on Grid'5000 [15]. The most recent cluster, *nova*[3], has been used for our experiments. Each machine in the cluster is equipped with two Intel Xeon E5-2620 v4 CPUs, 8 cores/CPU, 64GB RAM and 600GB HDD. The machines are interconnected with 10 Gigabit Ethernet. The nodes run Debian Linux 8.0.0 (jessie). The latency and bandwidth between machines are emulated using `tcconfig`[4], a wrapper tool for the Linux Traffic-Control tool [35].

C. System Setup

We compare Nitro with three existing systems, namely BitTorrent, InterPlanetary File System (IPFS) and OpenStack Swift.

a) Nitro: The key/value backend storage of Nitro is implemented using Redis [14], an in-memory database. We configure Redis to use append-only log and to sync the changes to the disk every second to ensure data persistence. Our chunk scheduling algorithm is implemented on top of the min-cut max-flow algorithm provided by the `networkx` python library version 2.0. We choose a chunks size of 256KB which provides a good trade-off between higher compression ratio and minimal metadata overhead. Also, 256KB is the default chunk size for BitTorrent and IPFS.

[3]https://www.grid5000.fr/mediawiki/index.php/Lyon:Hardware#Nova
[4]http://tcconfig.readthedocs.io/en/latest/pages/introduction/index.html

TABLE II
EMULATED NETWORK PROPERTIES OF 11 AWS REGIONS

	Bandwidth (Mb/s)	Latency (ms)
mean	67.28	179.97
std	41.70	79.19
min	16.76	27.20
25%	40.75	125.63
50%	56.20	171.40
75%	71.21	227.60
max	212.20	359.24

b) BitTorrent: The open-source BitTorrent client `libtorrent` version 1.1.4[5] and opentracker[6] have been used as client and tracker in the experiments.

c) IPFS: The IPFS version 0.4.10[7] is used for evaluation. IPFS uses Leveldb[8], a key-value store, as its backend storage system.

d) OpenStack Swift: We perform our experiments with the *Ocata* version of Swift. The read and write affinity properties are configured according to the bandwidth between the data centers.

Network emulation. We emulate 11 AWS regions in our experiments, including Virginia, California, Oregon, Ireland, Frankfurt, Tokyo, Seoul, Singapore, Sydney, Mumbai, and São Paulo. Statistical description of the emulated network latency and bandwidth values between those data centers can be found in Table II. The actual values for bandwidth are taken from a recent work [13], whereas the latency values can be found online[9]. The minimum bandwidth is between Singapore and São Paulo while the maximum bandwidth is between California and Oregon. The minimum latency is between California and Oregon while the maximum latency is between São Paulo and Sydney.

In our experiments, each data center is represented by one machine. For BitTorrent, an additional machine is used to run the tracker with an emulated latency of 25ms and bandwidth of 100Mb/s to other machines. The images in the compressed format are used as input by BitTorrent and Swift, and in the raw format by IPFS and Nitro as they are going to be deduplicated (and compressed) later by each system.

VII. EVALUATION RESULTS

For the evaluation, we first evaluate Nitro internals. Then we evaluate Nitro against other VMI management systems.

A. Effectiveness of Nitro

We evaluate the effectiveness of Nitro in three ways: 1) we show the impact of the global deduplication; 2) we compare the network-aware scheduling with random scheduling; and 3) we evaluate the runtime of the the network-aware algorithm.

[5]http://libtorrent.org/
[6]https://erdgeist.org/arts/software/opentracker/
[7]https://github.com/ipfs/go-ipfs
[8]https://github.com/google/leveldb
[9]https://www.cloudping.co, November 2017

1) Global deduplication: As we have discussed, the strength of deduplication compared to other compression techniques is the detection of identical blocks of data (i.e., chunks) on the dataset level. Therefore, with a bigger dataset, we may obtain higher compression ratio, as the chances of finding identical blocks become higher. In Figure 4a, we notice the increase of locally available chunks by each newly requested image. The x-axis represents the size of the dataset and the y-axis shows the number of locally available chunks and the missing chunks when requesting the current VMI. The lower area in blue represents the sum of chunks that are found locally for the complete dataset, i.e., the save in the network cost. Note that the number of locally available chunks for each image could be slightly different depending on the order in which the images have been requested, but we can always see the same trend.

2) Advantages of network-aware scheduling: We evaluate the effectiveness of our network-aware chunk scheduler by first comparing it with the random chunk scheduler. In this experiment, each image in the dataset is initially added to three sites randomly in a sequential order and then requested from a fourth site. As the size of transferred data in both setups (network-aware and random) is the same, we present the provisioning time, i.e., network transfer time, for both schedulers in Figure 4b. Results show that the network-aware scheduling of Nitro reduces the average provisioning time by 38% compared to the random scheduler.

We also evaluate the impact of the number of replicas on the transfer times as shown in Figure 4c. The y-axis shows the normalized transfer time. With more replicas, the chunk scheduling problem has a larger solution space as the chunks to be pulled are available on multiple sites. Thus, it is possible to obtain better transfer time results with Nitro when the number of replicas is large. For example, when the number of replicas is three, Nitro reduces the average transfer time by 58% compared to the random scheduler. Whereas if the chunks are available in a single site, the algorithm has no choice other than pulling all missing chunks from a single site (as we can see in the first column of Figure 4c). In conclusion, having more sources of data can help to reduce the provisioning time.

3) Runtime of network-aware scheduling: The runtime of the scheduling algorithm depends on many factors, such as the number of sites, number of chunks and the distribution of the chunks on sites as it determines the number of Mega Chunks (MCs). We measure the runtime while increasing the number of sites when pulling 10,000 chunks (representing 2.5GB of data). We consider the worst-case scenario for chunks mapping, i.e., there is at least one chunk that can be found in each subset of sites. In this case, the number of mega chunks is $2^{|sites|} - 1$, as discussed in Section III. In Table III, we compare the runtime of the algorithm *with* and *without* the grouping optimization, i.e., Mega Chunks. The runtimes are reported in seconds and measured using a machine with Intel i5 CPU and 16GB RAM. Results show that the grouping optimization can greatly reduce the runtime of the network-aware scheduling algorithm by up to 99.6%.

TABLE III
SCHEDULING ALGORITHM RUNTIME (SECONDS)

Sites	Number of MC	Runtime with MC	Runtime w/o MC
2	3	0.016	153.648
4	15	0.048	267.057
6	63	0.185	426.606
8	255	0.973	298.053
10	1023	8.035	2065.434

B. Nitro vs. IPFS

We choose IPFS for comparison with Nitro as it shares some similar design principles as Nitro. IPFS uses content-addressable storage to store data with deduplication applied. It transfers data by exchanging chunks between peers. Also, similar to Nitro, IPFS does not use compression. However, IPFS uses a Distributed Hash Table (DHT) to locate the chunks, whereas Nitro maintains the global distribution of chunks.

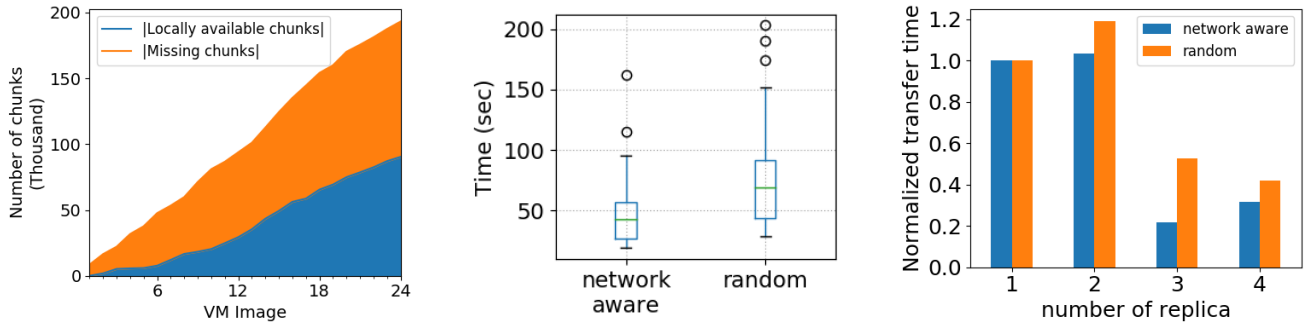
We compare the two systems in the same scenario as in the previous section with three replicas for each image. Figure 5a shows that Nitro reduces the network transfer time by 60% on average and 62% in the worst-case compared to IPFS. This can be explained by the fact that IPFS requests data from all available peers which have the data. As can be observed in Figure 5b, around 80GB of data is transferred across the network with IPFS, which is almost three times the size of the dataset. This technique is used by IPFS to tolerate network partitioning and weak links, but it results in huge network cost.

C. Nitro vs. Swift and BitTorrent

We further compare Nitro against two other systems, namely Swift and BitTorrent. We enable the chunk compression in Nitro as in the compared systems. Chunk compression leads to different chunks sizes. As a result, the network-aware scheduling of Nitro does not guarantee the optimal solution. However, we keep this setting for fair comparisons with the other two systems. Although BitTorrent divides images into pieces, it does not apply any kind of deduplication. On the other hand, Swift uses point-to-point replication and internally relies on `rsync`[10].

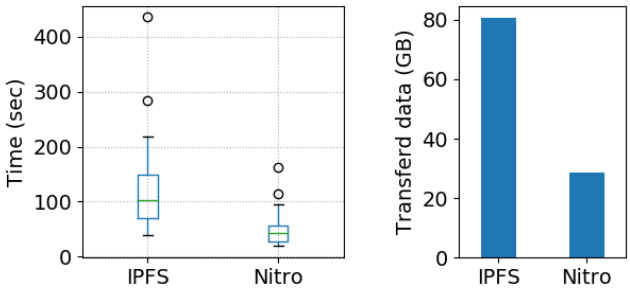
1) Provisioning time for single-site VMI request: In this scenario, we measure the transfer time for a single VMI request. Again, we set three replicas for each image and request each image from a fourth site. Figure 6a presents the results obtained by the studied systems. Nitro obtains the best network transfer time results. On average, it reduces the time by 77% compared to Swift and 46% compared to BitTorrent. Another observation is that, although Nitro does not guarantee optimal provisioning time for compressed chunks, the optimized VMI transfer time is still better than without compression when comparing Figure 6a with Figure 5a.

[10]https://docs.openstack.org/swift/latest/overview_replication.html#object-replication



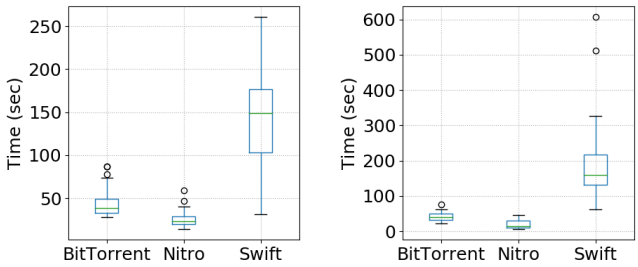
(a) Effectiveness of global deduplication (b) Effectiveness of network-aware scheduling (c) Replication sensitivity

Fig. 4. Evaluation results for Nitro internals: (a) Show the constant increase in redundant chunks (locally available) with the increase of dataset size (number of images) by using deduplication. (b) Compare the transfer time of network-aware scheduler and random scheduler. (c) Show how the transfer time of the two schedulers varies according to the number of initial number of replica.



(a) Transfer time (b) Network cost

Fig. 5. Comparison between Nitro and IPFS in terms of (a) total transfer time during the experiment and (b) total amount of transferred data over WAN.



(a) Single-site provisioning (b) Multi-sites parallel provisioning

Fig. 6. Comparison results of Nitro, Swift and BitTorrent: (a) The transfer time of the systems when requesting one VMI to one site. (b) The transfer time of the systems when requesting three VMIs to three sites, at the same time.

2) Provisioning time for multi-sites VMI request: In our design, Nitro provides chunk scheduling solution without being aware of the current status of the cloud system. Thus, it is possible to cause network contention when multiple sites are requesting VMIs at the same time. To evaluate how serious the network contention problem can get, we provision three VMs at the same time from three different sites, where five replicas of each image are available on the same sites. Figure 6b shows the obtained results. Nitro is still able to outperform the other two systems, with 53% and 90% reduction in the network transfer time compared to BitTorrent and Swift, respectively. This is mainly because the deduplication with compression in Nitro can reduce the size of chunks to be transferred across network compared to compression alone in Swift and BitTorrent (refer to Table I) by 56%. Moreover, the fact that Swift relies on point-to-point communication and cannot do the copy in parallel from different sites lies it behind the other two systems.

3) Sensitivity study on replication: The efficiency of peer-to-peer systems increases when the number of participating peers increases. Thus, we evaluate Nitro and BitTorrent when the participating sites, i.e., sites that have copies of the requested chunks, increase from three to five. Each image is initially uploaded to three random sites. Then, three different random sites request the image sequentially: the first site can request the image (chunks) from three locations, while the second and the third sites can request the chunks from four and five locations, respectively. We repeat the same steps for all the images. Figure 7 shows the normalized network transfer time results.

We have two observations. First, the network transfer times of both Nitro and BitTorrent decrease with the increase of number of replicas. This is consistent with our expectation. Second, Nitro outperforms BitTorrent in all cases, by up to 50%.

VIII. Conclusion

We introduce Nitro, a new VMI management system that is designed specifically for geographically-distributed clouds to achieve fast service provisioning. Provisioning services in geo-distributed clouds require transferring VMIs across the expensive and highly heterogeneous wide-area network. Different from existing VMI management systems, which ignore the network heterogeneity of WAN, Nitro incorporates two features to reduce the VMI transfer time across geo-distributed data centers. First, it makes use of deduplication

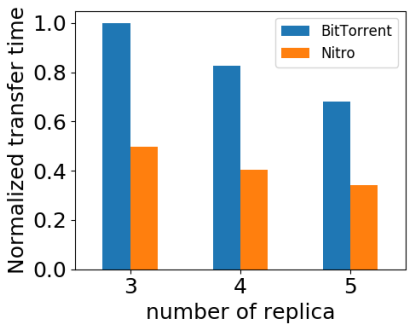


Fig. 7. Sensitivity study on replication for Nitro and BitTorrent when the data can be found in 3,4, and 5 sites.

to reduce the amount of data which will be transferred due to the high similarities within an image and in-between images. Second, Nitro is equipped with a network-aware data transfer strategy to effectively exploit links with high bandwidth when acquiring data and thus expedites the provisioning time. We evaluate Nitro by emulating real network topology. Results show that the network-aware data transfer strategy offers the optimal solution when acquiring VMIs while introducing minimal overhead. Moreover, Nitro outperforms state-of-the-art VMI storage systems (e.g., OpenStack Swift) by up to 77%. As future work, we plan to extend Nitro to work in the multi-site VMI request scenario.

ACKNOWLEDGMENT

This work is supported by the Inria Project Lab program Discovery, which is an Open-Science Initiative aiming at implementing a fully decentralized IaaS manager (see beyondtheclouds.github.io). It's also supported by the Shenzhen University Startup Grant (No.2018062). The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see www.grid5000.fr for details).

REFERENCES

[1] (2018, Feb.) Aws global infrastructure. [Online]. Available: aws.amazon.com/about-aws/global-infrastructure/
[2] (2018, Feb.) Windows azure regions. [Online]. Available: azure.microsoft.com/en-us/regions/
[3] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *SYSTOR*, 2009.
[4] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei, "An Empirical Analysis of Similarity in Virtual Machine Images," in *Middleware*, 2011.
[5] A. Karve and A. Kochut, "Redundancy Aware Virtual Disk Mobility for Cloud Computing," in *IEEE CLOUD*, 2013.
[6] A. Kochut and A. Karve, "Leveraging local image redundancy for efficient virtual machine provisioning," in *NOMS*, 2012.
[7] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *INFOCOM*, 2012.
[8] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein, "VMTorrent: Scalable P2P Virtual Machine Streaming," in *CoNEXT*, 2012.
[9] S. Bazarbayev, M. Hiltunen, K. Joshi, W. H. Sanders, and R. Schlichting, "Content-Based Scheduling of Virtual Machines (VMs) in the Cloud," in *ICDCS*, 2013.
[10] X. Xu, H. Jin, S. Wu, and Y. Wang, "Rethink the storage of virtual machine images in clouds," *Future Gener. Comput. Syst.*, 2015.
[11] B. Nicolae, A. Kochut, and A. Karve, "Discovering and Leveraging Content Similarity to Optimize Collective On-Demand Data Access to IaaS Cloud Storage," in *CCGrid*, 2015.
[12] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala, "Always Up-to-date: Scalable Offline Patching of VM Images in a Compute Cloud," in *ACSAC*, 2010.
[13] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds," in *NSDI*, 2017.
[14] (2018, Feb.) Redis. [Online]. Available: redis.io
[15] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*. Springer International Publishing, 2013.
[16] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST*, 2008.
[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010.
[18] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *OSDI*, 2006.
[19] (2018, Feb.) Openstack storage (swift). [Online]. Available: github.com/openstack/swift
[20] J. Benet, "IPFS - content addressed, versioned, P2P file system," *CoRR*, 2014. [Online]. Available: http://arxiv.org/abs/1407.3561
[21] B. Cohen. (2003) Incentives build robustness in bittorrent. [Online]. Available: http://www.bittorrent.org/bittorrentecon.pdf
[22] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," in *NSDI*, 2010.
[23] K. Oh, A. Chandra, and J. Weissman, "TripS: Automated Multi-tiered Data Placement in a Geo-distributed Cloud Environment," in *SYSTOR*, 2017.
[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS," in *SOSP*, 2011.
[25] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *SOSP*, 2013.
[26] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Perez, "Consistency in the Cloud: When Money Does Matter!" in *CCGrid*, 2013.
[27] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low Latency Geo-distributed Data Analytics," in *SIGCOMM*, 2015.
[28] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "WANalytics: Geo-Distributed Analytics for a Data Intensive World," in *SIGMOD*, 2015.
[29] A. C. Zhou, S. Ibrahim, and B. He, "On Achieving Efficient Data Transfer for Graph Processing in Geo-Distributed Datacenters," in *ICDCS*, 2017.
[30] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-Aware Optimization for Analytics Queries," in *OSDI*, 2016.
[31] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *SOSP*, 2009.
[32] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, Centralized Cluster Scheduling at Scale," in *OSDI*, 2016.
[33] R. K. Ahuja, M. Kodialam, A. K. Mishra, and J. B. Orlin, "Computational investigations of maximum flow algorithms," *European Journal of Operational Research*, 1997.
[34] (2018, Feb.) Vagrant. [Online]. Available: www.vagrantup.com
[35] (2006) Linux traffic control. [Online]. Available: tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html